

242ine

Issue #4

July 2017



Published by:

Timothy Swenson
swenson_t@sbcglobal.net
swensont@lanset.com

ZXzine is published as a service to the Sinclair ZX81 community. Writers are invited to submit articles for publication. Readers are invited to submit article ideas.

Created using Open
Source Tools:

- OpenOffice
- Scribus
- Gimp
- SZ81
- EightyOne

Copyright 2017
Timothy Swenson

Creative Commons License

- Attribution
- Non-Commercial
- Share-Alike

You are free:

- To copy, distribute, display, and perform the work.
- To make derivative works.
- To redistribute the work.

Editorial	1
Orbit	1
Random Number with Assembly	1
Partial Pascal	2
sz81 Emulator	3
Basic Computer Games	5
Saving Screen Blocks	5
Berch Assembler & Disassembler	6
z80dasm	8
Aardvark Adventures	9

Editorial

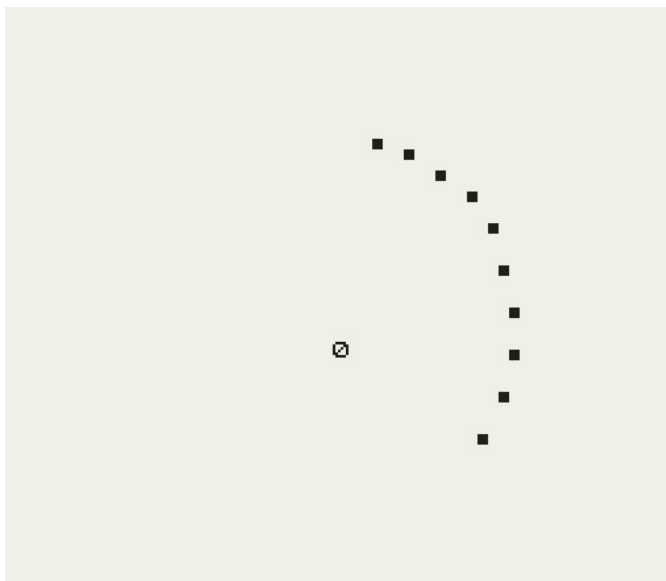
Slow and steady, that is how ZXzine comes along. It has not been a year since the last issue, but by only a few months. Luckily as I was putting this issue together some interesting article ideas came up, so I hope this issue will be worth the wait.

It's always interesting juggling working on the ZX81, the QL and other hobbies. It was Rob Heaton of the QL Forums that pinged me about "the next" issue and got me working on this issue. Like always there is a little history and some programming articles. It's really just those things that interest me on the ZX81.

Orbit

Back in college I wrote a program for a graphics class that I had picked up from a friend. It was a simple program that simulated an object orbiting around a planet or sun. I had originally written it for the QL then ported it to TurboPascal for the graphics class. I recently did a "fancier" version for the QL and thought I would do a version for the ZX81.

The program starts off with a planet or sun printed on the screen and a object (satellite or such) is represented by a single dot (or PLOT point). The object has the right velocity to orbit around the planet. The orbit is shown by a trail of dots. The trail is only 10 dots long and previous dots are



removed from the screen. This shows the orbit of the object, but does not clutter the screen with more and more dots.

The user can influence the object by using the arrow keys. Hitting the up arrow will give some thrust in the up direction to the object (up in relation to the viewer and not the sun/planet). Same goes for the other keys. With a little thrust in the right direction at the right time, the shape of the orbit can be made more elliptical. Be careful of using too much thrust as the object can be sent right out of orbit.

The program comes in both BASIC and C. The BASIC version is a little slow. The first version that I tried with C was too fast, so I added a delay to make it a little slower allowing the user time to put some input in. The non-slow C version is listed as orbit2.c.

Random Number with Assembly

Since having to come up with a random number generator with Smal-C on the QL, I've been interested in random number generators. When thinking about writing one program in assembly, I needed a random number generator. A quick Google search found a number. The problem was which one to use or in other words, which one was good enough to use. I decided to put them to the test.

I took a number of those that I found online, put a test program around them and had each one generate 100 random number in the range from 0-7 (an easy range with binary numbers). All of the generators created an 8-bit number (0-255), but to keep the tracking simple, use the SRL command a number of times to divide by two until I had a range between 0 and 7.

After running the test programs, I counted how many times each random number was generated. I created a spreadsheet with the output from 6 different generators. My first generator did not pass the test, as it really did not generate a truly random set of numbers. So, I did the spreadsheet with #2 through #5.

The spreadsheet shows the results. The standard deviation value shows how far apart the count is between the rolled numbers. In theory, count for

	A	B	C	D	E
1		RND2	RND3	RND4	RND5
2	0	16	16	17	13
3	1	11	16	10	11
4	2	16	14	14	16
5	3	10	9	16	18
6	4	14	13	8	9
7	5	14	12	10	14
8	6	11	10	14	10
9	7	8	10	11	9
10	Total	100	100	100	100
11	AVG	12.5	12.5	12.5	12.5
12	STD Dev	2.927700219	2.725540575	3.207134903	3.338091842
13					
14					

each number rolled should be equal. If I let the test program go for a thousand iterations, the numbers might be closer, but to keep it simple, I stayed with just 100 iterations. The lower the standard deviation is the better the random number generator.

The standard deviation for RND3 was the best at 2.72 with RND2 coming second at 2.92. RND4 and RND5 are less random, but they are shorter and would work if memory is tight. I did not test for speed, but I don't think any of the random number generators are slow enough to cause a processing issue.

Partial Pascal

In looking to see what other languages are available for the ZX81, I ran across one Partial Pascal, probably the only native structured programming language for the ZX81. It was written, of all places, Wheaton, Illinois, by Mike Amling. Partial Pascal can be found from a number of download sites. It is sometimes listed as "Pookah Partial Pascal". I'm guessing Pookah is the person that converted it from tape to .p format. Even though the software is available, it really is not usable because the manual is missing. All that I know about Partial Pascal comes from Ads, articles or tinkering with the software.

Partial Pascal has three sections; an editor, the compiler, and executing the programs. Since most

ZX81's did not have a disk drive, Partial Pascal saved the Pascal code to tape, to be read in later by the compiler. The compiler then saved the finished executable to tape, which was then read in by the part of the program to execute the code.

As far as I know, this was the only program and/or language on the ZX81 that did this. Most of the others save snapshots of ZX81 memory with all of the data (like Vu-Calculator).

Partial Pascal. as the name implies, does not support the full Pascal language. From the

advertisement for Partial Pascal, it had the following language features:

IF... THEN .. ELSE....
CASE statement
FOR loops
FUNCTION
PROCEDURE

It does lack:
records, set,
label, goto
and reals.

The system is listed a "device independent" meaning that data can be written to the screen, the printer, and even tape. Data can also be read from the tape to the program. It would be interesting to know if a simple text

Partial Pascal

Structured Programming

Partial Pascal's IF is a full IF condition THEN one or more statements with optional ELSE one or more statements. The CASE statement selects among many alternatives. Programs can loop by testing a condition at the top of a loop, testing at the bottom or bumping a variable using FOR TO. FUNCTIONS and PROCEDURES (subroutines) can have parameters, their own temporary variables and their own subroutines.

Device Independence

Partial Pascal programs can write data to tape, just like to the screen or printer. And data on tape can be read back in by any Partial Pascal program.

Full-screen Editor

The editor gives full cursor control over a 22-line window into your program or data. Commands include insert/delete character, move window up/down, insert/delete line (no line numbers, so a new line can go anywhere), save, load or merge from tape, and more.

The Partial Pascal programming package includes editor, compiler, example programs, run-time interpreter and user manual. Partial Pascal is a subject of ISO Pascal without record, set, label, goto and reals.

16K ZX81 or TS1000 reqd. \$30 postpaid
Semper Software
1569 Brittany Court Wheaton, IL 60187

file with numeric data could be created with the text editor and then read in by a Pascal program.

The compiler has a full screen editor with 22 lines for coding. The advertisement lists the different commands that can be run, so it might be a mode-ed editor, like VI, where editing and running commands happen at different times. The advertisement mentions that you can "save, load or merge from tape."

Once the code is written, it is passed through the compiler, then it can be read in again to be executed. The advertisement lists the whole package as "The Partial Pascal programming package includes editor, compiler, example programs, run-time interpreter and user manual."



What is interesting is that Pascal was originally written to be used with 9-track tape drives, which were pretty standard devices in the computer room when Pascal was written. By the 80's, Pascal had moved from tape systems to disk systems. Here was Pascal being taken back to a tape system.

The only reference that I can find of the compiler being used is by Edward Snow, who wrote an article for the North American Timex/Sinclair User Group newsletter on benchmarking. He used Basic, Forth and Pascal to see how each did on the ZX81. He actually published the source code to his benchmarking program. The article was published in the Spring of 1992.

It took a lot of work to get a Pascal parser, tokenizer and compiler to fit on the ZX81. Given the

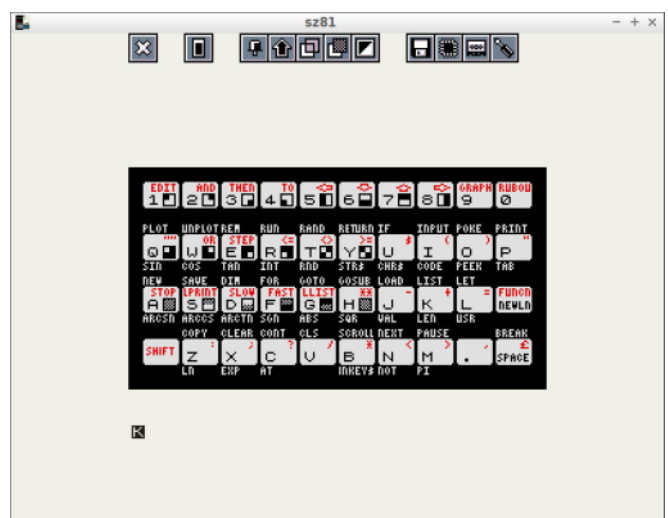
compilers reliance on the tape system, even with the manual, I don't think the compiler would work on the newer emulators.

In a series of photographs from the 1987 Timex Sinclair Computer Fest with Mike Amling of Semper Software at his booth, looking completely bored. It sounds like most of the attendees did not realize the significance of getting Pascal to run on a 16K system.

sz81 emulator

When I was running Windows on my laptop, I used EightyOne as my ZX81 emulator. Moving to Linux required that I find a new emulator that was native to Linux. After searching online, I found sz81 an emulator that is based on Ian Collier's xz80 and Russell Marks's z81.

sz81 comes as a source code release in a tar.gz file. It requires a compiler to compile the program and GCC is the recommendation. Another requirement is the SDL (Simple DirectMedia Layer) graphics library. I downloaded the sz81 tar.gz file, unzipped, untared and ran the make command to compile it. Once compiled, there is a sz81 binary file in the source directory. To start the emulator I created a



desktop shortcut on my desktop, using the icon that was in package.

sz81 starts quickly and takes a few seconds to get to the K cursor. The default configuration is the ZX81, but that can be changed to the ZX80 if needed. Clicking a mouse anywhere in the

emulator window will bring up the menu and a virtual keyboard. The menu is used for controlling the emulator, like doing a reset, loading .P files, changing the hardware configuration, etc. The virtual keyboard is an exact copy of the ZX81



keyboard. It can be used as a reminder what keys do what or the mouse can be used to click on the keys like a real keyboard. I hardly type programs in the emulator, and use tools that create .P files. I usually just have to hit R for RUN to start the programs after loading them.

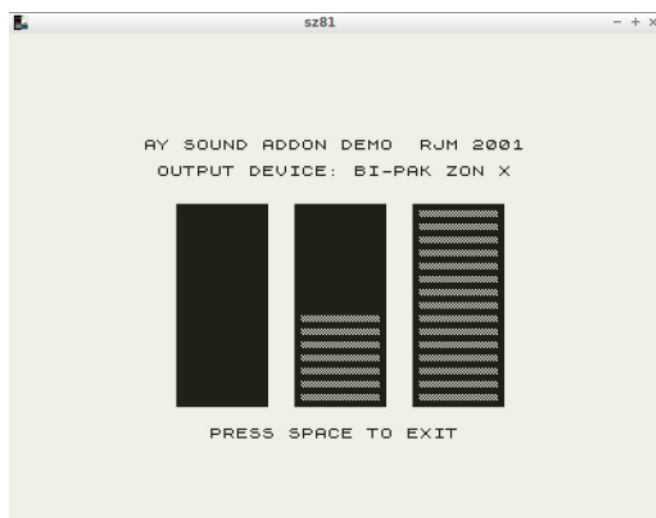


Loading .P files is simple with a mouse driven menu system that is used to change directories and drill down to the location of the .P files. sz81 is installed in its own directory and I keep my ZX81 files in a different directory, so I have to go up one level, and then down the “zx81” directory to get to the files I want to load.

One limitation I found with sz81 was the inability to do hi-res graphics. The version that I was using was 2.1.7. For a long time that was the latest

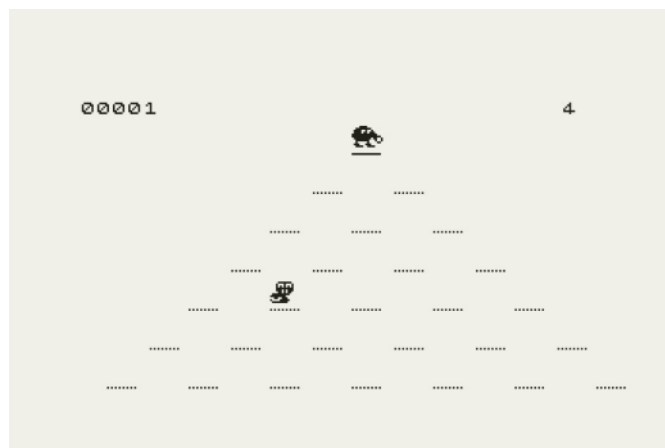
version available. I then found that E. Olofsen had picked up the development of sz81 and came out with version 2.2.0, which does support hi-res graphics. The most recent release is 2.3.6, but I think I’ll stay with 2.2.0 for a while.

sz81 does come with some demo binaries, including a large number of 1K hi-res games. One interesting demo is an audio demo using the Bi Pak Zon X-81 programmable sound generator (PSG) emulation. It shows how three channel sound can be used. The source code for the demo. Doing a Google search I found an online version of the manual for the Zon X-81. I had heard of the PSG but never bought it or had seen it. When I ran the sound demo I was



surprised by the three channel sound.

sz81 also comes with the Open81 ROM, which is a source code release version of the T/S 1000 or improved ZX81 ROM. To create a ROM from the Open81 ASM file, Pasmu is needed. I have been using Pasmu for a while, so I just compiled the Open81 ROM and used it instead of the default zx81.rom file. So far I have yet to see any issues.



There are a number of features of sz81 that I have yet to tinker with. I just wanted a good solid, quick, ZX81 emulator for Linux and sz81 has met the bill. The latest versions of sz81 can be found at:

<http://rullf2.xs4all.nl/sz81/>

Basic Computer Games

The ZX81 was first publicly advertized in the October 1981 edition of Popular Science. I was a senior in high school and had been looking to buy a computer and had some funds saved up. I quickly sent in the order for the ZX81 and received it in late November.

Since the ZX81 was so new to the market, there were no books available at any bookstore. The three computer shops in town did not carry the ZX81 so they did not carry any books. What I did find at my local

bookstore was "Basic Computer Games" by David Ahl, the Microcomputer edition, published in 1978. The book is a collection of short computer games written in

Basic. The first edition of the book was published by Digital Equipment Corp. (DEC) in 1973 and written in a variety of Basic dialects. The Microcomputer version had the games written for Microsoft Basic.

Microsoft Basic was the version of Basic on the Altair and was similar to the Basic for the Apple II. The book listed ways to convert the programs for Basic on the Tandy Model 1, SWTPC 6800,

Processor Technology SOL, IMSAI, Ohio Scientific, and Northstar Horizon. Most of these systems were S-100 bus-based and ran a version of CP/M.

The programs were tested on an Altair and a Model 43 Teletype machine was used for generating the printouts and the program examples. The programs were all of the type that were originally designed to be run on a teletype machine, where all output was sent to a printer.

There is 101 Basic games in the book, from Acey Duecy to Word. Some games were classic games, like Checkers, Hangman, Blackjack, and Reverse. Some were computer only games, like Life, Mugwump, Slalom, and Super Star Trek. The book was also illustrated with robots acting out the games. The illustrations kept the book interesting to look at and the games made the book interesting to read and understand what is going on.

The book was useful in that it taught me some Basic techniques that were not in the ZX81 Users Guide. I learned a lot from porting the programs to the ZX81, changing them to work with graphics. I would use the games as a start of a program idea. I would then use what I liked from the game code, adding what I needed to make it run on the ZX81. If I had a different way of doing something, I would use my way.

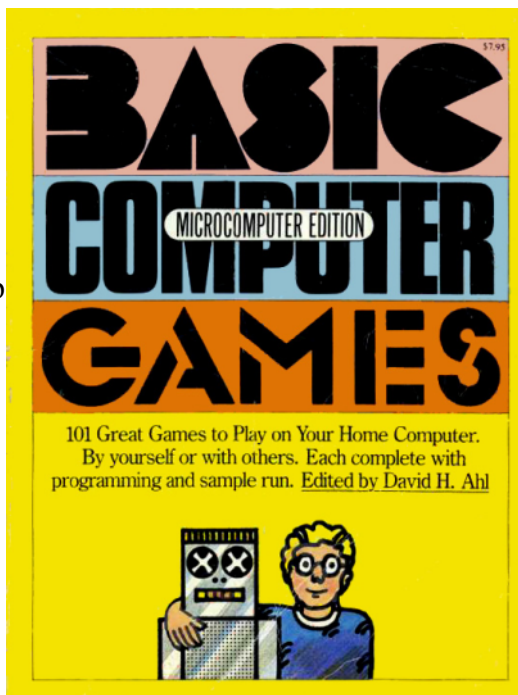
The book is now a nice nostalgic read, taking me back more than 35 years. The book is available from the following link:

https://annarchive.com/files/Basic_Computer_Games_Microcomputer_Edition.pdf

Saving Screen Blocks

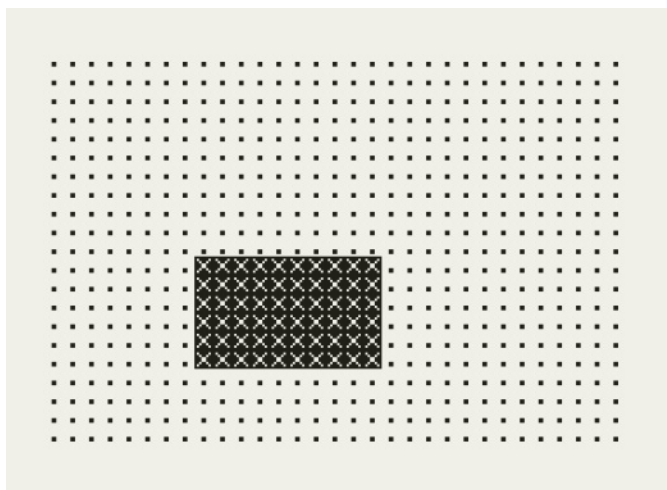
The ZX81 has a very simple screen memory layout. It is text only and each location on the screen is a single byte. This makes it easy to understand and easy to manipulate.

With the concept of windows on other computers, I wondered about saving parts to the screen and then restoring it after a period of time. I came up with



two routines, saveblk and returnblk, to save a block of screen memory and then restoring it back to the screen. To keep things simple, the blocks are saved to an array. Each routine takes a set of coordinates, a starting X,Y screen location and an ending X,Y screen location. The program does not check to confirm that the saveblk or returnblk routines are given the same arguments, it is up to the user to make sure of that. Saveblk read the bytes in memory from that area of the screen and saves it to an array. Returnblk takes the data in the array and copies it to the location in screen memory. The routines know where the location of screen memory is via the system variable DFILE.

If the dimensions of the starting X,Y and ending X,Y are kept, it is possible to use the routines to move a block of screen memory from one location to another. Since the array is not cleared after using returnblk, it is also possible to make copies of the block around the screen.



```
/* saves block of screen memory */
saveblk(x1, y1, x2, y2)
    unsigned int x1, y1, x2, y2;
{
    unsigned int x, y;
    unsigned char z;

    for (x = x1; x <= x2; x++) {
        for (y = y1; y <= y2; y++) {
            z = bpeek(dfile + (x * 33) + y
+ 1);
            m[(x*MAX_X)+y] = z;
        }
    }
}

/* restores block of screen memory */
returnblk(x1, y1, x2, y2)
    unsigned int x1, y1, x2, y2;
{
    unsigned int x, y;
    unsigned char z;

    for (x = x1; x <= x2; x++) {
        for (y = y1; y <= y2; y++) {
            z = m[(x*MAX_X)+y];
            bpoke((dfile + (x * 33) + y +
1), z);
        }
    }
}
```

The first version was done in Basic making easy to test the routine. The second version was written in C with Z88DK. Z88DK has two built-in functions called bpeek and bpoke, short for byte peek and byte poke. The functions are exactly like PEEK and POKE in Basic. Using the functions saves the trouble of using pointers to address memory directly.

It would take more work to allow more than one part of the screen to be saved at one time and would likely need to use a malloc call in C to allocate memory for each call the

routines. Something for another day.

Berch Assembler & Disassembler

Most of the software for the ZX81 originated in the UK and was brought to the US. I was always interested in our home grown software. One that I remember seeing advertised is an assembler and disassembler from Bob Berch, of Rochester, New York.

Luckily, someone archived Berch's assembler, both the software and the manual, and made them available online. I thought it would be interesting to give the program a try and see how easy it is to use. I've included the manual and the software in the distribution zip file for issue.


```
ZX ASSEMBLER/DISASSEMBLER
V1.0 FOR 28-32K
COPYRIGHT 1983 BOB BERCH
```

```
* EOF 0000
```

The program is the type that sits above RAMTOP, so when a new program is loaded, it is still available to be used. The assembly code is written in REM statements. The code looks like this:

```
30 REM LD A,38
```

```
10 REM ;XXXXXXXXXXXXX
20 REM ; TEST OF BERCH ASSEMBL
ER
30 REM LD A,38
40 REM CALL H0010 ; CALL PRINT
50 REM RET
60 REM END
```

```
0/0
```

Any comments must be preceded by a semi-colon (;). The comments can be on a line by themselves or they can come after the assembly code. For finished code to be placed in the first REM statement, the REM statement must first be created.

I decided to write my code outside of the ZX81 and then use zxtxt2p to convert the text file to a .P file. I added line numbers, but really did not need to. The test code I used is test.bas.

I loaded the program into my ZX81 emulator. An introductory screen was shown and the program was then loaded into higher memory. I hit Shift-Q to exit the program. I then loaded my test.p program, making sure to use the version of LOAD that does not clear all of memory (including above

RAMTOP). I LISTed my program to confirm that it had loaded correctly.

To start the assembler, I ran the command RAND USR 30000. The assembly came up showing a disassembly of lower memory. I hit Z to assembly

```
4406 3E 26 00 LD A,26
4408 CD 10 00 CALL 0010
440B C9 RET
440C 80 ADD A,B
440D 6A LD L,D
440E 60 LD H,B
440F 00 NOP
4410 00 NOP
4411 76 HALT
4412 8F ADC A,A
4413 6A LD L,D
4414 60 LD H,B
4415 00 NOP
4416 00 NOP
4417 00 NOP
4418 00 NOP
4419 00 NOP
441A 00 NOP
441B 00 NOP
441C 00 NOP
441D 00 NOP
441E 00 NOP
* EOF 440B
```

my code, then Y to confirm. The program was then showing on the screen in memory (along with the memory after my code).

To move the finished code to the REM statement, I hit the M key (for move) and entered 4082. The program wants a 4 digit memory location, so it must be entered in hex. I then did Shift-Q to exit the assembler, listed my program and noticed that the REM statement had changed. I deleted all of the other lines but the REM statement. I added in the line:

```
20 RAND USR 16514
```

I then saved the program. When I loaded the program, I would hit RUN and it would execute.

For really long programs I can see where deleting the code in the REM statements could take a little time.

The next thing I wanted to do was to test the disassembler part of the program. For this, I wanted to use a assembly program that I had compiled with Pasm0. Disassembly was far easier to do than the assembly. I loaded the program, then loaded the program that I compiled with Pasm0. It did the call to the assembly program. The next step was to point the disassembler to the area in memory

where my code as. I hit A and then entered 4082. Up came the disassembly of my code. At first it looked odd, because it was in ZX81 character code, but I just had to hit the D key to toggle from Data to Disassembly. What was interesting was that the code that I wrote in assembly was not at 4082, but stored a few bytes further. It looks like Pasmo had one assembly line and then a call to where my code was.

```

4082 7E      LD A,(HL)
4083 C3 89 40 JP 4089
4086 00     NOP
4087 00     NOP
4088 76     HALT
4089 3E 26   LD A,26
408B CD 10 00 CALL 0010
408E C9     RET
408F 00     NOP
4090 01 0B 00 LD BC,000B
4093 F9     LD SP,HL
4094 D4 C5 0B CALL NC,0BC5
4097 1D     DEC E
4098 22 21 1D LD (1D21),HL
409B 20 0B   JR NZ,40A8
409D 76     HALT
409E 76     HALT
409F 20 1C   JR NZ,40BD
40A1 24     INC H
40A2 1E 00   LD E,00
40A4 00     NOP
40A5 23     INC HL
*                               EOF 0000

```

I don't know if I would use the Berch assembler over something like Pasmo, but it was interesting to use it and gives the programmer another option. If someone is a die-hard user of only the original hardware, then the Berch assembler would do well.

z80dasm

For writing assembly language programs, I use Pasmo, which is a cross-compiler running on Linux that takes the ASM code and converts it into a .P file. After tinkering with the Berch assembler and disassembler, I wondered if there was a cross-disassembler that I could use under Linux. After I Google search, I found z80dasm.

z80dasm is available for Debian-based systems and installs with the command:

```
sudo apt-get install z80dasm
```

If need by the source code can be found on github or from the link:

<https://www.tablix.org/~avian/z80dasm/>

The gzipped tar file for version 1.1.3 is only 111K in size.

After installing it, I started reading the manual. Basically, you give z80dasm a binary file and it will disassemble from the start of the file. In the case of a .P file, the executable code is not at the start of the file and there is a number of bytes that need to be skipped.

z80dasm has a block file that is used to tell z80dasm about blocks of the file and to treat them as something other than code. For the .P file, I created a block file with the following entry:

```
; z80dasm 1.1.3
; command line: z80dasm -g 16385 -b
block.txt -o test.out test.P
```

```

org 0412dh

defb 000h
defb 000h
.....
.....
defb 000h
defb 076h
ld a,026h
call 00010h
ret
nop
ld bc,0000bh
ld sp,hl
call nc,00bc5h
.....
.....

```

header: start 16385
end 16513 type
bytedata

This, combined with a command line argument telling z80dasm that the start of the .P file is the same as ORG 16385, z80dasm will list the data before the executable code as data. This all assumes that the executable code is stored in a REM statement.

Since it is hard to tell when the executable code ends, I am unable to create a

block file that tells z80dasm when to stop disassembling.

The command line to use with this block file is:

```
z80dasm -g 16385 -b block.txt -o file.out file.P
```

The -o option is for setting the output file. If left out, then the output is sent to STDOUT. The last argument is the .P file to do the disassembly on.

I have tried z80dasm on a second .P file using the same block file and the results were the same, the disassembly starting at 16514.

Aardvark Adventures

Aardvark Technical Services was a company founded in April 1981 by Roger Olsen, in Walled Lake, Michigan. The company started selling computer adventure games for a number of computer systems. An advertisement from 1981 listed the following computer systems; OSI, TRS-80, TRS-80 Color, Sinclair, PET, and VIC-20. They would later create games from the Apple II and C64. The company was advertised as "Aardvark - The Adventure Place"

Aardvark ran ads in a number of magazines for different computer systems, including Sync magazine, the number one magazine in the US for Sinclair computers. A number of the ads that I remember had the artwork for the game Quest, big and bold at the top of the ad.

Their ads were full of text describing each game. The types of adventure games they had were; Quest, Mars, Pyramid, Earthquake, Derelict, Haunted House, and so on. Their ads said that all of the games were available for all of the computer systems. The prices were \$14.95 for games on cassette and \$19.95 for disk version.

Bob Retelle was one of the authors that wrote games for Aardvark. He said that Aardvark was not known for the quality of its games. He said "I remember discovering some of the bugs of other

games in early testing (it was pretty common for Rodger to hand out tapes of new games before they were put in the catalog), but it was like pulling teeth to get him to fix any of them."

Unlike some other game companies that found ways to make the games portable across systems, all of the Aardvark adventure games were written in the native BASIC for each system. As for porting the games to different systems, Bob mentions "... sometimes the original authors did the conversions, sometimes it was high school kids hired to come into Aardvark after school who did it."

Later, Aardvark expanded into video games for the different systems. In the 1983 company catalog, there is "Zart Invaders" for the Timex/Sinclair 1000, written in machine code by G. Lamon. Seawolf, also written in machine code for the C64, Vic-20 and Tandy Color Computer.

The company did not survive the early computer days and wrapped up business in May 1986.

I never did order any games from Aardvark, but their ads were everywhere and very iconic for the early days of home computing.

